



Compilers and Code Optimization

EDOARDO FUSELLA

Contents

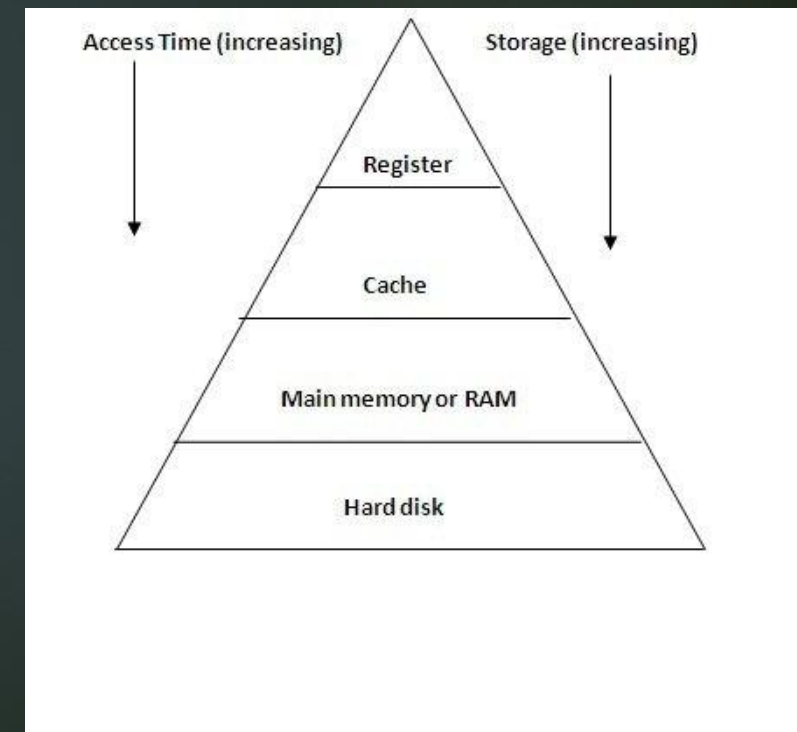
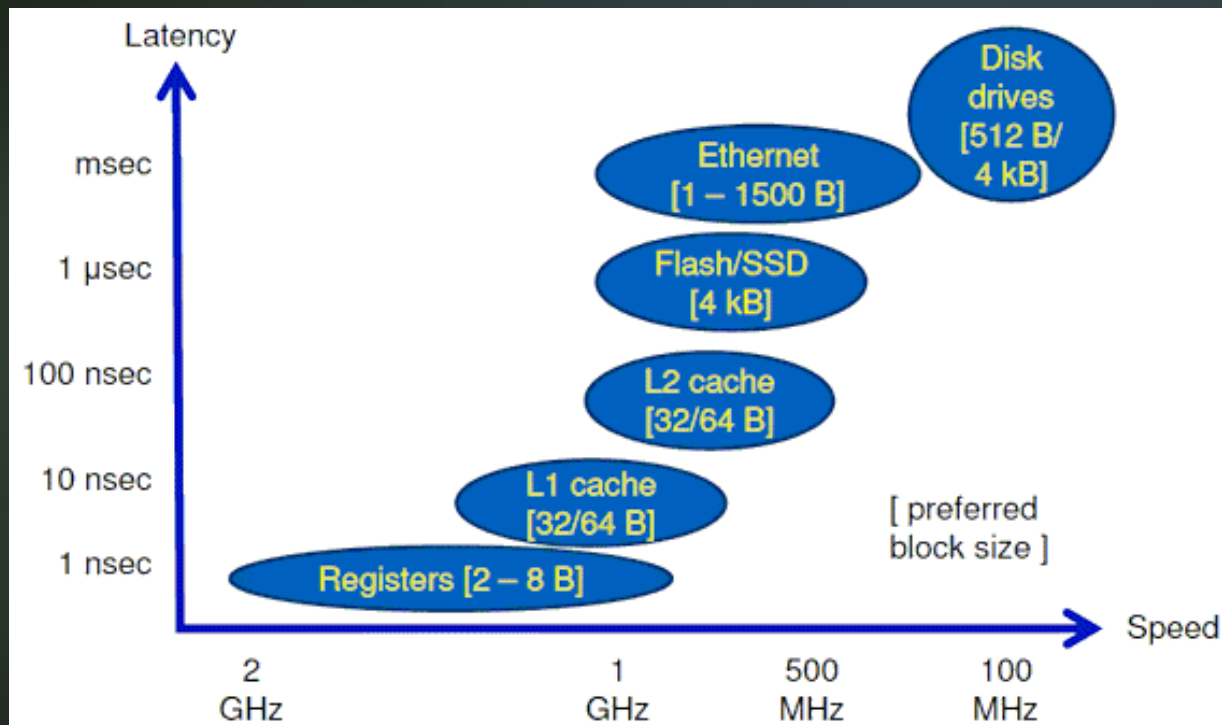
- ▶ Data memory layout
- ▶ Instruction selection
- ▶ Register allocation



Data memory layout

Memory Hierarchy

► Capacity vs access speed



Main memory

- ▶ Classes of objects
 - ▶ Stack resident: the variables and arguments of subprograms
 - ▶ Heap resident: created by *new* or *malloc*, reachable from a stack or heap-resident variable by pointers
- ▶ Memory Management
 - ▶ Data to memory mapping for stack resident objects: static
 - ▶ Data to memory mapping for heap resident objects: dynamic

Memory for subprogram data

Goals

- ▶ In almost any modern programming language, a function may have local variables that are created upon entry to the function.
- ▶ Several invocations of the function may exist at the same time, and each invocation has its own instantiations of local variables.
- ▶ Memory allocated to each activation of a procedure/function/method
- ▶ We focus on procedural/imperative languages.

Memory for subprogram data

Example

```
int f(int x) {  
    int y = x+x;  
    if (y<10)  
        return f(y);  
    else  
        return y-1;  
}
```

- ▶ A new instantiation of x is created (and initialized by f 's caller) each time that f is called
- ▶ Because there are recursive calls, many of these x 's exist simultaneously.
- ▶ Similarly, a new instantiation of y is created each time the body of f is entered.

Memory for subprogram data

Why a stack?

- ▶ In many languages (including C, Pascal, and Java), local variables are destroyed when a function returns.
- ▶ Since a function returns only after all the functions it has called have returned, function calls behave in last-in-first-out (LIFO) fashion.
- ▶ If local variables are created on function entry and destroyed on function exit, then we can use a LIFO data structure – a stack – to hold them.

Memory for subprogram data

Generalities

- ▶ Different invocations of the same subprogram need separate data areas
- ▶ The data memory layout is the same for all activations of the same subprogram
- ▶ The traditional memory organization is a push-down stack of frames
- ▶ Modern compilers use CPU registers instead of data areas for storing parameters, data and return address
- ▶ Mirror memory locations are still required in many cases (e.g., number of available registers not enough, recursive procedure calls, etc.).

Data Memory Layout

Activation records of subprograms

Frame or activation record

- ▶ The memory area used for the data of a subprogram (function, procedure, method)
- ▶ Frames of invoked and not yet terminated subprograms are managed as elements of a push-down stack
- ▶ The most recently invoked subprogram has its frame on top of the stack
- ▶ The frame of the main program is located at the bottom of the stack
- ▶ The stack used for frames allows not only push and pop, but also access to intermediate cells (it is an array)
 - ▶ When local variables are created they are not always initialized right away
 - ▶ After many variables have been pushed, we want to continue accessing variables within the stack

Data Memory Layout

Activation records of subprograms

Frame or activation record

- ▶ A memory segment delimited by two addresses:
 - ▶ Stack pointer: sp points to the end of current stack frame
 - ▶ Frame pointer: fp points to the end of previous stack frame, i.e. to the base of the current one.
- ▶ At least sp is kept in register for faster access

Data Memory Layout

Activation records of subprograms

Fixed or variable frame size

- ▶ $|sp - fp|$ is the frame size
- ▶ For C the frame size is known when the subprogram is compiled
- ▶ i.e., all invocations of a given subprogram have equal frame size
 - ▶ The stack usually grows only at the entry to a function, by an increment large enough to hold all the local variables for that function, and, just before the exit from the function, shrinks by the same amount.
 - ▶ Not true for all languages (e.g., variable size arrays in stack)
- ▶ All locations beyond the stack pointer are considered to be garbage
- ▶ All locations before the stack pointer are considered to be allocated

Data Memory Layout

Activation records of subprograms

Frame allocation and layout

- ▶ On procedure invocation a frame for the invoked subprogram (termed *callee*) is allocated next to the frame of the *caller*
- ▶ The *incoming arguments*, i.e. the actual parameters of the current subprogram, are included in the frame of the caller
- ▶ Frame layout can be different depending on the compiler
- ▶ However, to allow cross-language interoperability, a standardized frame layout is needed
- ▶ Recommended frame layouts may be based on ISA properties

Data Memory Layout

Activation records of subprograms

Frame contents

- ▶ Local variables: i.e. variables whose lifetime coincides with the life of the current subprogram
- ▶ Temporaries: intermediate results of expressions computed in the subprogram
- ▶ Outgoing arguments: Actual parameters of a subprogram called inside the current one.
- ▶ Saved registers: space for saving register values.
- ▶ Bookkeeping information: for frame and instruction linkage on returns from subprogram.

Data Memory Layout

Frame layout

- ▶ For historical reasons, stacks usually start at a high memory address and grow toward smaller addresses.
- ▶ Stacks grow downward and shrink upward
- ▶ The frame has
 - ▶ *Incoming arguments* passed by the caller
 - ▶ Part of the previous frame but at a known offset from the frame pointer
 - ▶ *return address* created by the CALL instruction
 - ▶ *local variables* kept in the frame
 - ▶ Sometimes local variables kept in a register needs to be saved into the frame to make room for other uses of the register
 - ▶ *outgoing argument* space to pass parameters

<i>comments</i>	<i>stack of frames</i>	<i>comments</i> ↑ higher addresses
incoming arguments	argument <i>n</i> argument 2 argument 1	frame of caller
frame pointer →		
	local variables	
	return address [dynamic link] temporaries saved registers	frame of current subprogram
out-going arguments	argument <i>m</i> argument 2 argument 1	
stack pointer →		next frame ↓ lower addresses

Data Memory Layout

Frame pointer

- ▶ With many procedures open at a time, where to store frame pointers?
- ▶ Fixed frame size: fp computed at return ($fp = sp + fsize$)
- ▶ Parametric frame size: place dynamic link in each frame holding fp during subprocedure calls
- ▶ In languages allowing statically nested subprograms, more bookkeeping is needed for enforcing scope rules

Data Memory Layout

Frame pointer

- ▶ A function $g(\dots)$ calls the function $f(a_1, \dots, a_n)$
- ▶ On entry to f
 - ▶ the stack pointer points to the first argument that g passes to f
 - ▶ f allocates a frame by simply subtracting the frame size from the stack pointer sp
 - ▶ The old sp becomes the current frame pointer fp
 - ▶ The old value of fp is saved in memory (in the frame)
- ▶ When f exits
 - ▶ It copies fp back to sp
 - ▶ It fetches back the saved fp

Data Memory Layout

Return address

- ▶ *ra* address of the instruction to be executed when the activation ends
- ▶ In source/intermediate program, the instruction following the call
- ▶ Refers to the memory segment where code is stored (possibly read-only memory)
- ▶ When the current procedure is done, $pc \leftarrow ra$
- ▶ Jump-to-subroutine instructions typically save *ra* in a register
- ▶ Saving *ra* not necessary if a procedure is a leaf

Data Memory Layout

Addresses of frame resident variables

Compiler actions

- ▶ Assign frame positions to arguments and local variables
- ▶ Add epilogue/prologue code to create/destroy frame at runtime

Sample function

```
int f(int x) {  
    int y = 3*x + m;  
    if (y>10)  
        return g(y)+y;  
    else  
        return y+2;  
}
```

Data Memory Layout

Addresses of frame resident variables

Frame of sample function

		↑ higher addresses
incoming argument	x	size 2 bytes
frame pointer →		
local variable	y	size 2 bytes
	return address	
	temporaries	
	saved registers	frame of f
outgoing argument	f	
stack pointer →		
		↓ lower addresses

Data Memory Layout

Addresses of frame resident variables

- ▶ A variable in the frame has a relative address or offset with respect to the fp
- ▶ For instance, x is $fp + 2$, y is at offset 0 from fp , and ra is at offset -2
- ▶ Global variables reside at known offsets from the bottom of the stack

Data Memory Layout

Need of frame pointer

In a fixed frame size language

- ▶ The address of every frame resident item can be expressed as an offset from the stack pointer sp
- ▶ Using only sp : save instructions at return, save one register
- ▶ Number of temporary cells is not known when the front-end finishes
- ▶ Solution: use explicit fp or move computation of frame size after register allocation

Data Memory Layout

Use of registers for arguments

- ▶ Modern CPUs have a moderate number of registers
- ▶ Registers can be used for storing arguments, making calls faster
- ▶ Usually limited to 4-6 arguments (enough for most calls)
- ▶ Mirror memory locations are needed for recursive procedures or just deep call stacks

Who saves the registers in a procedure call?

- ▶ A function *f* uses *r1* for variable *y*, then it calls functions *g*, which also needs *r1*
- ▶ The value of *y* in register *r1* must be stored into a stack frame, before *g* uses *r1*
- ▶ Then, when *g* returns to *f*, the saved value must be loaded into *r1*

Data Memory Layout

Use of registers for arguments

Who saves the registers in a procedure call?

- ▶ caller-save the caller saves the register value before the call and restores it after the call
- ▶ callee-save the register is preserved across procedure calls by the callee
- ▶ Note that this decision is generally specified by the manufacturer/designer
- ▶ Optimization:
 - ▶ If f knows that the value of some variable x will not be needed after the call, it may put x in a caller-save register and not save it when calling g
 - ▶ if f has a local variable i that is needed before and after several function calls, it may put i in some callee-save register ri , and save ri just once (upon entry to f) and fetch it back just once (before returning from f)

Data Memory Layout

Use of registers for arguments

Need of an address:

- ▶ a variable whose address is taken (using &)
- ▶ a variable that is passed by reference
- ▶ A variable that is accessed from a nested function, not in C but in other languages
- ▶ The variable is an array, for which address arithmetic is necessary to extract components
- ▶ The value is too big to fit into a single register
- ▶ There are so many local variables and temporary values that they won't all fit in registers, in which case some of them are “spilled” into the frame

In all such cases the variable is said to escape and must be allocated in the frame.

Data Memory Layout

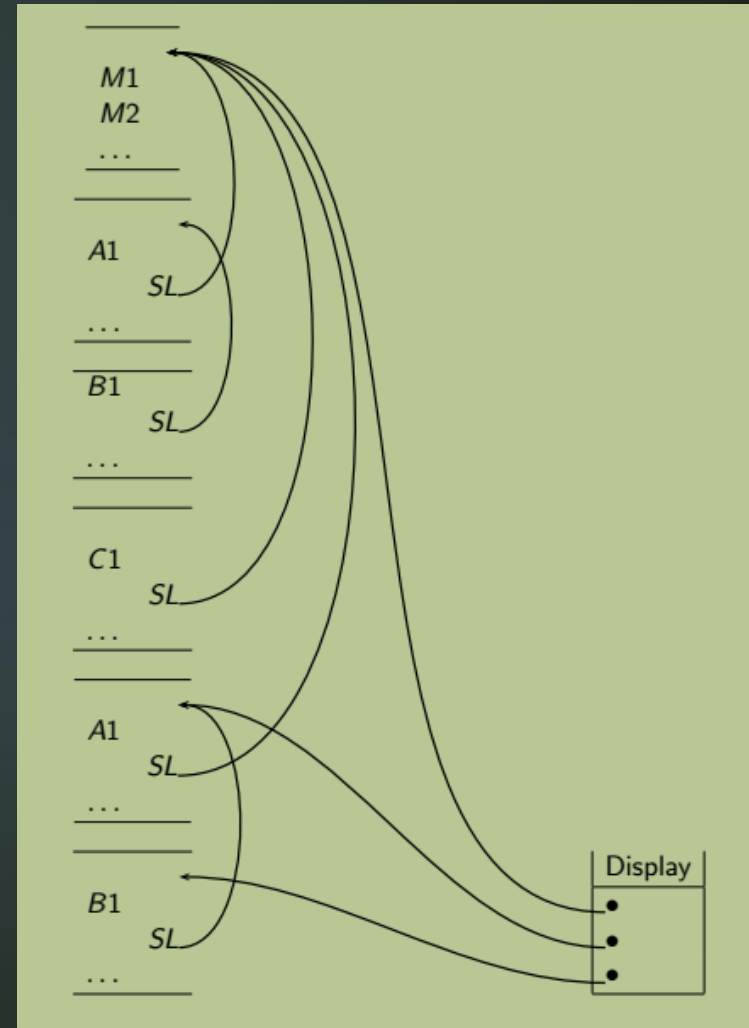
Frame Management for Nested Procedures

- ▶ In languages that allow nested function declarations (such as Pascal, and Java), the inner functions may use variables declared in outer functions.
 - ▶ Nested procedures access the frame of their parent
- ▶ Static link: whenever a subprogram f is called, it can be passed an extra argument, i.e. a pointer to the frame of the most recently entered subprogram statically enclosing f . This pointer is termed static link, and has a fixed offset in the frame.
 - ▶ Deep nesting leads to chains of references
 - ▶ To reduce overhead, a global display array can be maintained
- ▶ Lambda lifting: When g calls f , each variable of g that is actually needed by f (including by any subprogram nested inside f) is passed to f as an extra argument

Data Memory Layout

Frame Management for Nested Procedures

- ▶ Access a non-local variable x
 - ▶ load the static link sl into a register $r1$
 - ▶ $\text{addr}(x) = r1 - \text{offset}(x)$
 - ▶ load the memory value into a register
 - ▶ more memory accesses needed for distant scopes





Instruction selection

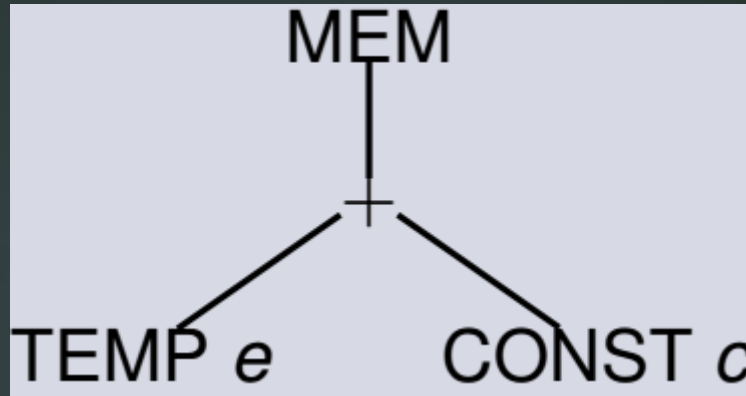
Instruction Selection

Overview

- ▶ Translate IR to machine instructions
- ▶ Historically, code generators were hand written
- ▶ Need for retargetability
- ▶ Need for complex code generation in CISC machines
- ▶ Pattern matching algorithms

Instruction Selection

Example



This is typical of taking the value of a frame resident variable at offset c from the base, with e the frame pointer.

Machine code translations:

<i>poor</i>	<i>better</i>	<i>best</i>
$r_0 \leftarrow c$	$r_1 \leftarrow e + c$	$r_2 \leftarrow M[e + c]$
$r_1 \leftarrow e + r_0$	$r_2 \leftarrow M[r_1]$	
$r_2 \leftarrow M[r_1]$		


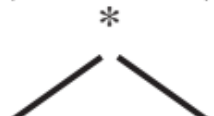
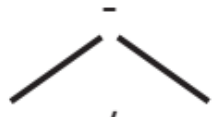
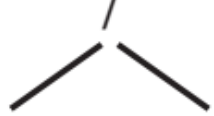
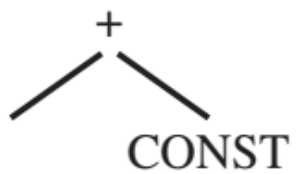


Instruction Selection

Tree Patterns

- ▶ We can express a machine instruction as a fragment of an IR tree, called a *tree pattern*.
- ▶ Instruction selection is the task of tiling the tree with a set of tree patterns.
- ▶ The tiles are the set of tree patterns corresponding to legal machine instructions, and the goal is to cover the tree with nonoverlapping tiles.

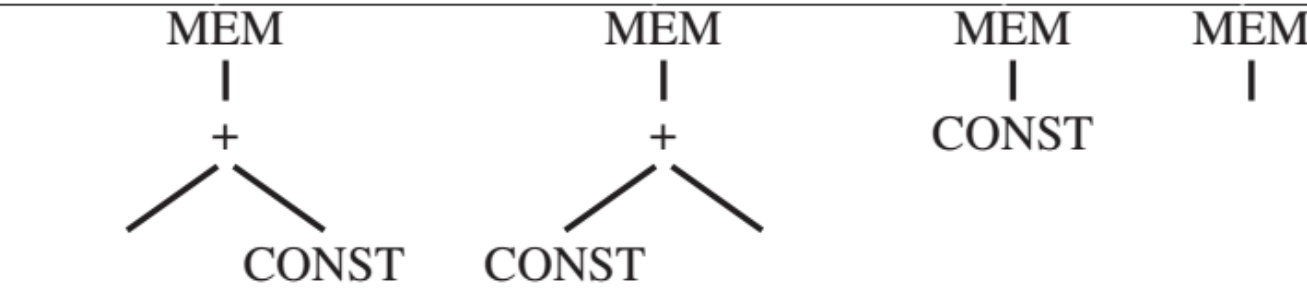
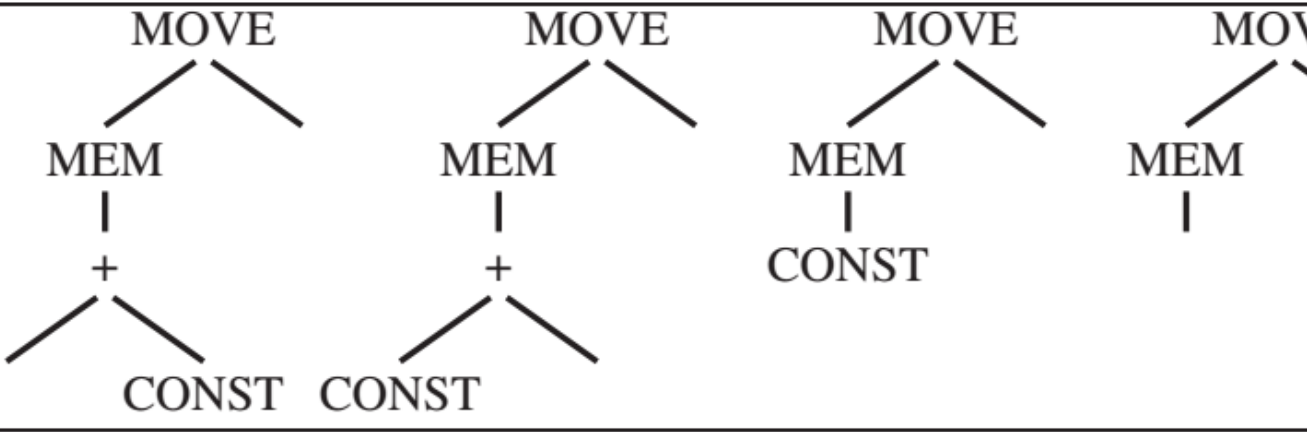
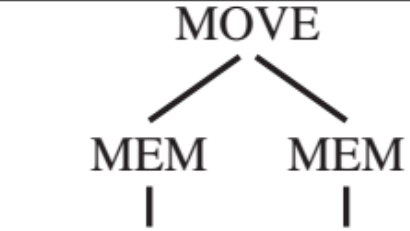
Instruction Selection

Example Target

<i>Name</i>	<i>Effect</i>	<i>Trees</i>
—	r_i	TEMP
ADD	$r_i \leftarrow r_j + r_k$	
MUL	$r_i \leftarrow r_j \times r_k$	
SUB	$r_i \leftarrow r_j - r_k$	
DIV	$r_i \leftarrow r_j / r_k$	
ADDI	$r_i \leftarrow r_j + c$	  CONST
SUBI	$r_i \leftarrow r_j - c$	 CONST

Instruction Selection

Example Target

<i>Name</i>	<i>Effect</i>	<i>Trees</i>
LOAD	$r_i \leftarrow M[r_j + c]$	
STORE	$M[r_j + c] \leftarrow r_i$	
MOVEM	$M[r_j] \leftarrow M[r_i]$	

Optimal IR tree covering

Goal of the Code Generator

- ▶ To find a set of instruction tiles such that:
 - ▶ all the nodes of the IR tree are covered;
 - ▶ the set has near to minimum cost.

What is optimal? Cost criteria

- ▶ Execution time: sum of the execution times of all instructions (in clock cycles) — very rough estimate unless pipelining and other architectural issues are taken into account
- ▶ Energy: often proportional to instruction latency/execution time

Tree covering example

Preliminaries

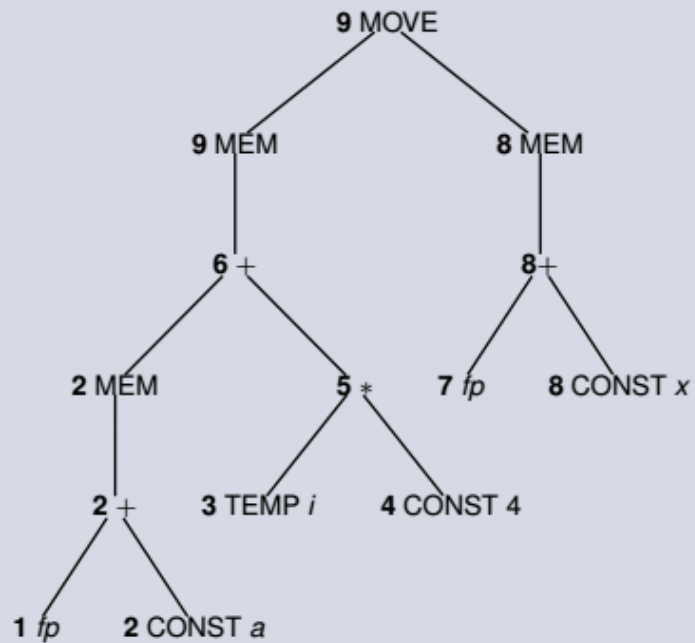
IR tree of $a[i] = x$, where

- ▶ a is the base address of the pointer to an array
- ▶ 4 bytes the array element size
- ▶ x is a frame resident variable

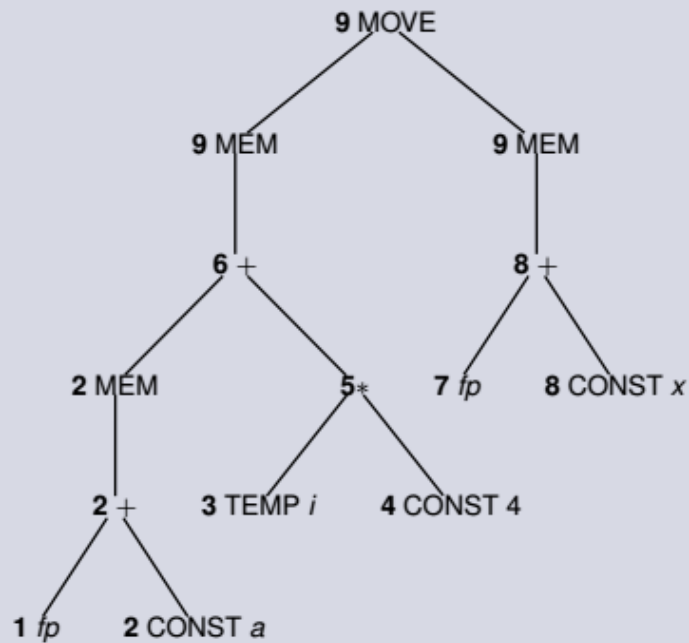
Assuming a cost of 1 for all instructions except MOVEM, which costs 2, we get the following coverings

Tree covering example

Two possible coverings



2 LOAD $r_1 \leftarrow M[fp + a]$
4 ADDI $r_2 \leftarrow r_0 + 4$
5 MUL $r_2 \leftarrow r_i \times r_2$
6 ADD $r_1 \leftarrow r_1 + r_2$
8 LOAD $r_2 \leftarrow M[fp + x]$
9 STORE $M[r_1 + 0] \leftarrow r_2$
cost = 6



2 LOAD $r_1 \leftarrow M[fp + a]$
4 ADDI $r_2 \leftarrow r_0 + 4$
5 MUL $r_2 \leftarrow r_i \times r_2$
6 ADD $r_1 \leftarrow r_1 + r_2$
8 ADDI $r_2 \leftarrow fp + x$
9 MOVEM $M[r_1] \leftarrow M[r_2]$
cost = 7

- ▶ r_0 always contains zero
- ▶ a and x are frame-resident

Maximal Munch

Top-down Tree Covering

Selection Algorithm (assuming equal costs)

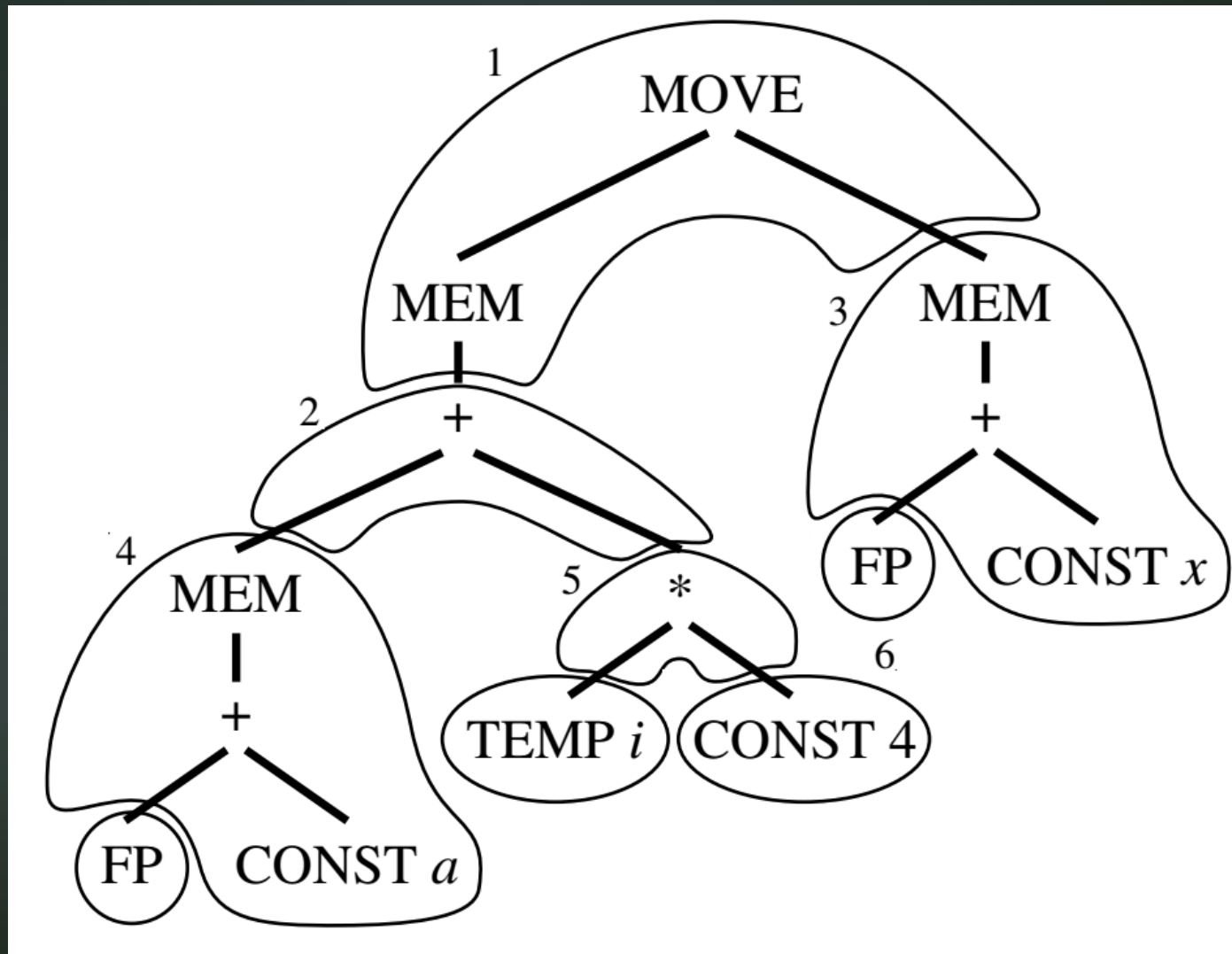
1. Consider the root O of the tree
2. $\text{munch}(O)$: choose a largest instruction tile matching O and perhaps several other nodes near the root
3. Let $O_1; O_2; \dots; O_k$ be the roots of the subtrees which border with the covered chunk
4. Recursively munch the trees $O_1; O_2; \dots; O_k$ (in this order)
5. Terminate when the entire tree has been munched

Code Generation

- ▶ While the algorithm visits the tree in depth-first order, it appends the tiles to a list
- ▶ To produce the actual code, the list is reversed
- ▶ The algorithm is fast and operates in linear time, w.r.t. the tree size

Maximal Munch

Example



Dynamic Programming

Maximal Munch

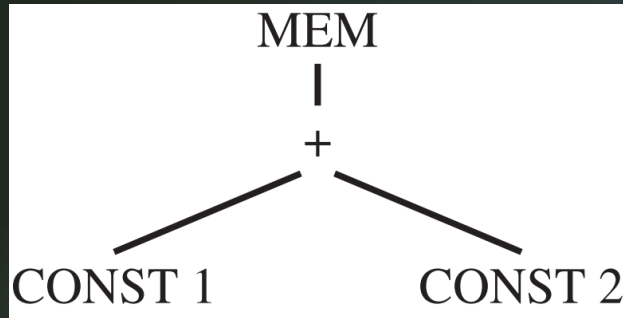
- ▶ Optimum solution in case of all instructions with the same cost

Dynamic Programming

- ▶ It is possible to improve the quality of the solution by moving to a Dynamic Programming algorithm
- ▶ The dynamic programming algorithm works bottom-up, building the most efficient solution for increasingly larger subtrees
- ▶ A cost to every node in the tree equals to the sum of the instruction costs of the best instruction sequence that can tile the subtree rooted at that node.
- ▶ After building the solution for the root node, a pre-order visit of the tree is sufficient to perform the Code Generation

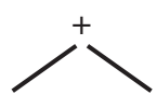

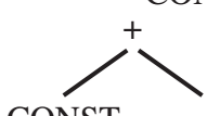
Dynamic Programming


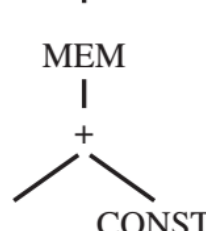
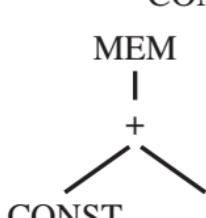
An example



- ▶ The only tile that matches CONST is an ADDI instruction with cost 1.
- ▶ Several tiles match the + node
- ▶ Several tiles match the MEM node
- ▶ Cost of the root node is 2

ADDI $r_1 \leftarrow r_0 + 1$
LOAD $r_1 \leftarrow M[r_1 + 2]$

Tile	Instruction	Tile Cost	Leaves Cost	Total Cost
	ADD	1	1+1	3
	ADDI	1	1	2
	ADDI	1	1	2

Tile	Instruction	Tile Cost	Leaves Cost	Total Cost
	LOAD	1	2	3
	LOAD	1	1	2
	LOAD	1	1	2

Code Generation for CISC machines

RISC vs CISC

A typical modern RISC machine

- ▶ 32 registers
- ▶ only one class of integer/pointer registers
- ▶ arithmetic operations only between registers
- ▶ “three-address” instructions of the form $r1 \leftarrow r2 \oplus r3$
- ▶ load and store instructions with only the $M[reg + const]$ addressing mode
- ▶ every instruction exactly 32 bits long
- ▶ one result or effect per instruction

A typical CISC machine

- ▶ few registers (16, or 8, or 6)
- ▶ registers divided into different classes, with some operations available only on certain registers
- ▶ arithmetic operations can access registers or memory
- ▶ “two-address” instructions of the form $r1 \leftarrow r1 \oplus r2$
- ▶ several different addressing modes
- ▶ variable-length instructions, formed from variable-length opcode plus variable length addressing modes;
- ▶ instructions with side effects such as “autoincrement” addressing modes.

Code Generation for CISC machines

CISC example: the Pentium architecture

The Pentium, in 32-bit mode

- ▶ six general-purpose registers, a stack pointer, and a frame pointer
- ▶ most instructions can operate on all six registers, but the multiply and divide instructions operate only on the *eax* register
- ▶ “two-address” instructions, meaning that the destination register must be the same as the first source register
- ▶ most instructions can have
 - ▶ two register operands: $r1 \leftarrow r1 \oplus r2$
 - ▶ one register and one memory operand: $M[r1 + c] \leftarrow M[r1 + c] \oplus r2$

Code Generation for CISC machines

instruction selection for the Pentium architecture

1. Few registers

- ▶ We continue to generate TEMP nodes freely, and assume that the register allocator will do a good job.

2. Classes of registers

- ▶ The multiply instruction on the Pentium requires that its left operand (and therefore destination) must be the *eax* register.
- ▶ The highorder bits of the result are put into register *edx*.
- ▶ How to implement $t_1 \leftarrow t_2 \times t_3$? The solution is to move the operands and result explicitly.

<code>mov eax, t₂</code>	$eax \leftarrow t_2$
<code>mul t₃</code>	$eax \leftarrow eax \times t_3; \quad edx \leftarrow \textit{garbage}$
<code>mov t₁, eax</code>	$t_1 \leftarrow eax$

Code Generation for CISC machines

instruction selection for the Pentium architecture

3. Two-address instructions

- ▶ How to implement $t_1 \leftarrow t_2 + t_3$? By adding extra move instructions

<code>mov t_1, t_2</code>	$t_1 \leftarrow t_2$
<code>add t_1, t_3</code>	$t_1 \leftarrow t_1 + t_3$

4. Arithmetic operations can address memory (memory-mode operands)

- ▶ Fetch all the operands into registers before operating and store them back to memory afterwards.
- ▶ These two sequences compute the same thing:

1 cycle each	<code>mov eax, [ebp - 8]</code>	<code>add [ebp - 8], ecx</code>	3 cycles , less registers
	<code>add eax, ecx</code>		
	<code>mov [ebp - 8], eax</code>		

Register allocation

Register Allocation

Introduction

Motivation

- ▶ Access to registers much faster than access to memory
- ▶ The mapping from variables to registers is many-to-one
 - ▶ Previous phases of the compiler assume that there are an infinite number of registers to hold temporary values and that MOVE instructions cost nothing
- ▶ The number of live variables may exceed available registers
- ▶ Variables must be saved into memory to free needed registers

Goal

- ▶ To assign the many temporaries to a small number of machine registers, and, where possible, to assign the source and destination of a MOVE to the same register

Scope

1. For expressions such as $(x + 8 \times y) - z = (y + 1)$
2. For basic blocks
3. For one procedure at a time
4. For the entire program

Register Allocation

Introduction

Advantages of small allocation units

- ▶ Easily integrated with the Instruction Selection phase
- ▶ Simpler and faster
- ▶ More suitable for application in dynamic compilation
 - ▶ Global management of registers across procedures can increase the code performance, but is more complex for the compiler
- ▶ We focus on intra-procedural Register Allocation

Register Allocation

Liveness Interference

Principles

- ▶ If two variables are both live in the same program point, they cannot be stored in the same register
- ▶ Use two registers or store one or both in memory
- ▶ If the liveness intervals of two variables are disjoint, the same register can be used

Liveness interference relation

- ▶ A binary, symmetric relation between two variables a and b denoting the fact that the liveness intervals of a and b overlap. The interference relation can be depicted as a non-directed graph.

Register Allocation

Graph Coloring

How to allocate registers?

- ▶ The Liveness Interference relation can be represented as a graph
- ▶ Registers can be viewed as colors
- ▶ The assignment needs to paint nodes of the graph with colors, so that adjacent nodes differ in color
- ▶ Similar problems: coloring of political maps

Graph coloring problem

- ▶ Not always solvable for given number of colors
- ▶ Solution: spill variables to memory and recompute liveness
- ▶ Graph coloring is intractable!
 - ▶ One of Karp's 21 NP-complete problems
- ▶ Use practical ($O(n)$) heuristics

Register Allocation

Graph Coloring: Main Steps

Repeat until successful:

Build Construct Liveness Interference Graph G

Repeat until \exists nodes of degree less than K :

Simplify Reduce the graph removing colorable nodes

If $G \neq \emptyset$:

Spill Select from remaining nodes a candidate for spilling

Go to Simplify

Select Assign registers or mark nodes as spilled

If not succeeded :

Start over Modify code to manage spills

- ▶ adding store/load instructions for the nodes we are not able to find a color

- ▶ Denote with K the number of available registers/colors
- ▶ The algorithm employs a stack S

Register Allocation

Graph Coloring

Simplify

- ▶ If G contains a node m with fewer than K neighbors
- ▶ Construct the graph $G' = G - \{m\}$
- ▶ Push m on S
 - ▶ If G' can be colored, so does G
 - ▶ Worst case: $K - 1$ colors are needed for m 's neighbors, thus leaving a color for m .
 - ▶ If the graph is empty at the end, go to Select, else go to Spill

Register Allocation

Graph Coloring

Spill

- ▶ If $G \neq \emptyset$, all remaining nodes have $\geq K$ neighbors
- ▶ Choose a node s and mark it as a *potential candidate for spilling*
- ▶ Push s on S and compute $G' = G - \{s\}$
- ▶ Retry **Simplify**

Register Allocation

Graph Coloring

Select

- ▶ For each node $s \in S$, assign a color and pop it
- ▶ If s does not interfere with the previous popped node, reuse the same color
- ▶ As long as s is not marked for spilling, it is guaranteed to be colorable
- ▶ If s is a spilling candidate, either the node can be colored, or the potential spill is marked as an actual spill
- ▶ If there are actual spills, apply them in the code and Start over

How could a potential spill s not turn into an actual spill?

Register Allocation

Graph Coloring: an example

- ▶ four registers ($K = 4$)
- ▶ live on entry: $k; j$
- ▶ live on return: $d; k; j$

$g := \text{mem}[j + 12]$

$h := k - 1$

$f := g * h$

$e := \text{mem}[j + 8]$

$m := \text{mem}[j + 16]$

$b := \text{mem}[f]$

$c := e + 8$

$d := c$

$k := m * 4$

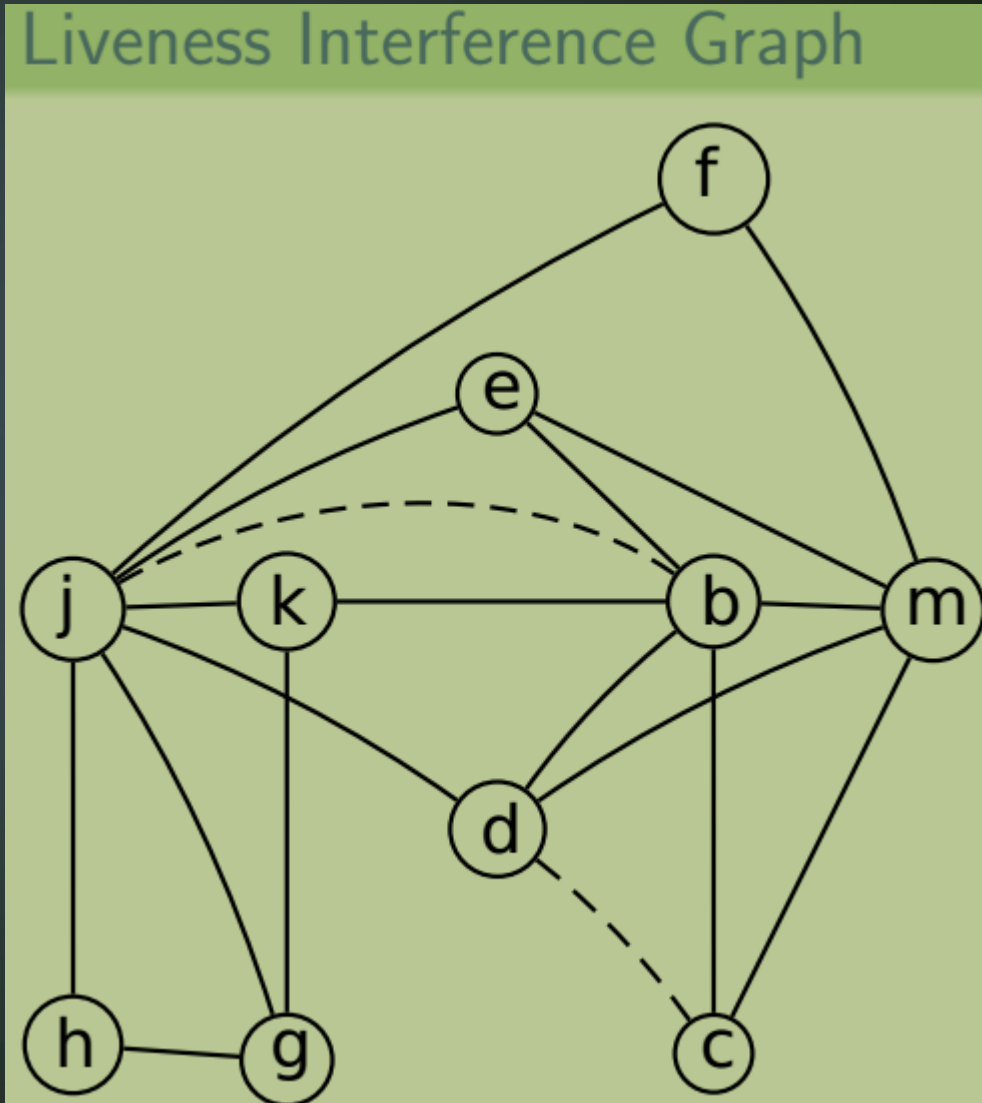
$j := b$

– move instruction

– move instruction

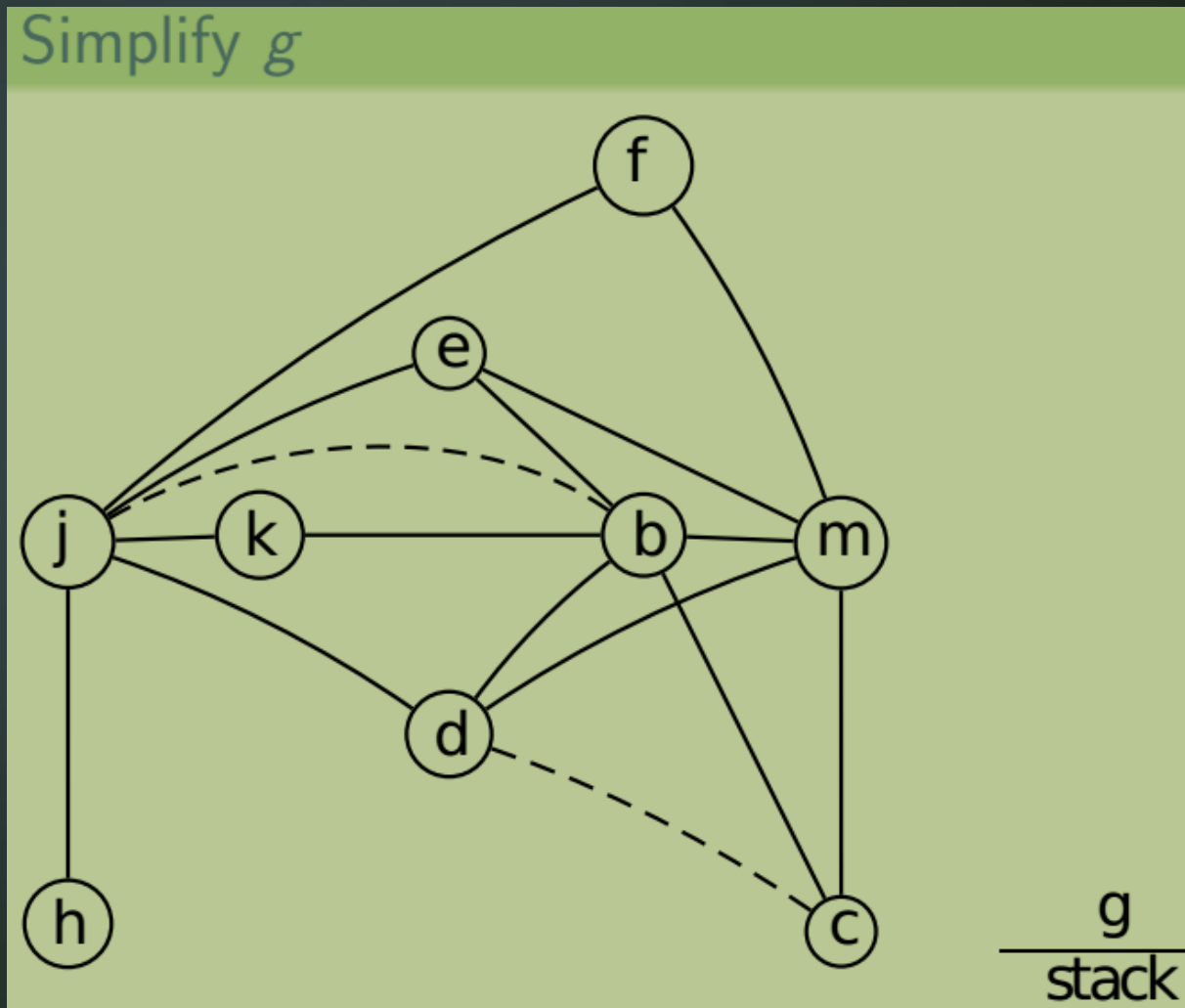
Register Allocation

Graph Coloring: an example



Register Allocation

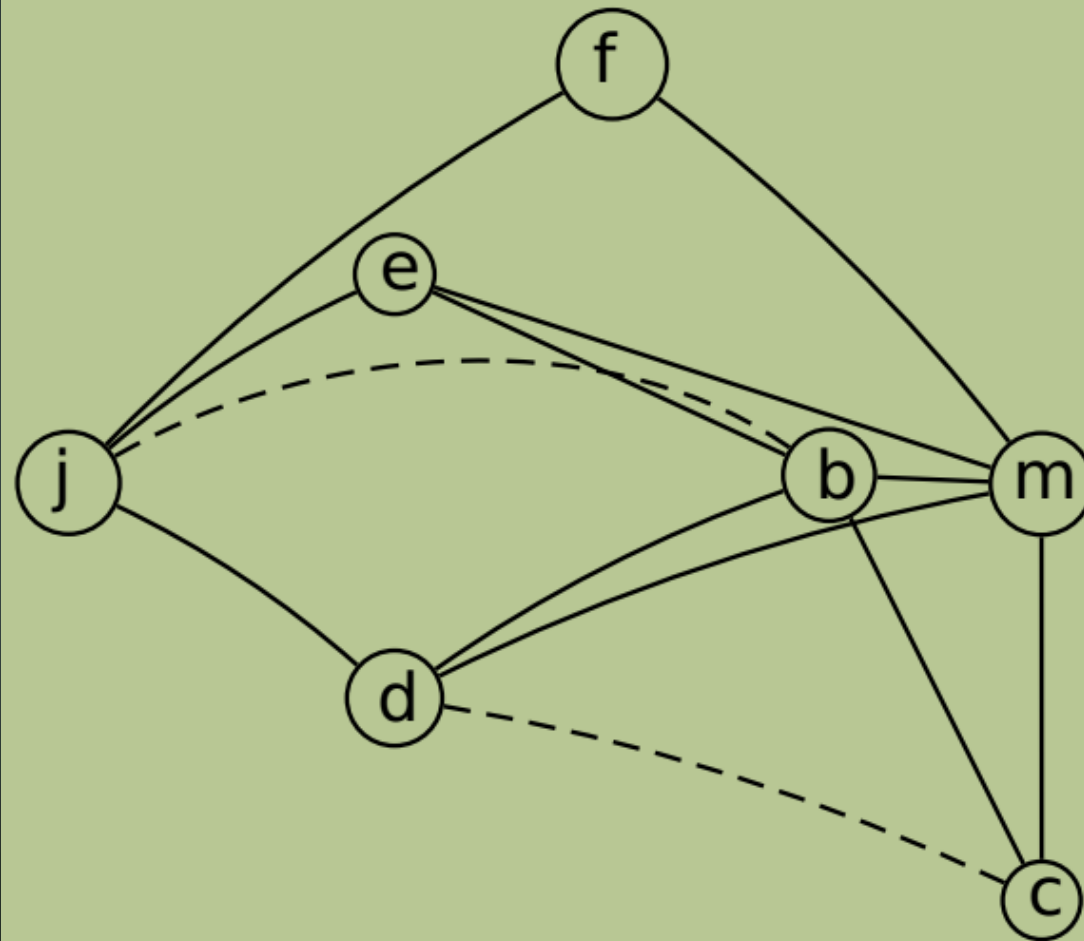
Graph Coloring: an example



Register Allocation

Graph Coloring: an example

Simplify h, k

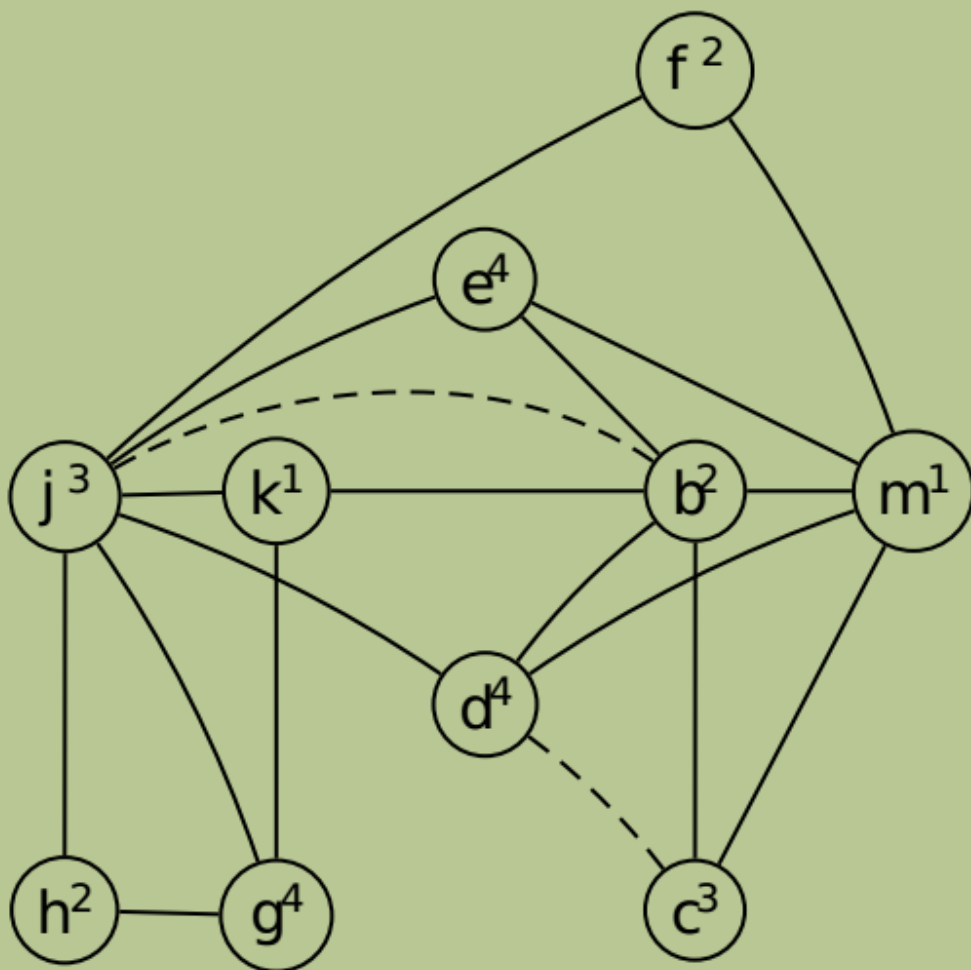


k
h
g
<hr/>
stack

Register Allocation

Graph Coloring: an example

Coloring



m	1
c	3
b	2
f	2
e	4
j	3
d	4
k	1
h	2
g	4
<hr/>	
stack	assignment

Register Allocation

Coalescing

How to eliminate redundant move instructions?

- ▶ If there is no edge in the interference graph between the source and destination of a move instruction, then the move can be eliminated.
- ▶ The source and destination nodes are coalesced into a new node whose edges are the union of those of the nodes being replaced.
 - ▶ In principle, any pair of nodes not connected by an interference edge could be coalesced.
- ▶ Unfortunately, the node being introduced is more constrained than those being removed, as it contains a union of edges.
 - ▶ A graph, colorable with K colors before coalescing, may no longer be K -colorable after reckless coalescing.
- ▶ We wish to coalesce only where it is safe to do so, that is, where the coalescing will not render the graph uncolorable

Register Allocation

Coalescing: Briggs strategy

- ▶ Nodes a and b can be coalesced if the resulting node ab will have fewer than K neighbors of significant degree (i.e., having $\geq K$ edges).
- ▶ The coalescing is guaranteed not to turn a K -colorable graph into a non- K -colorable graph
 - ▶ Simplify will remove all the insignificant degree nodes from the graph
 - ▶ The coalesced node will be adjacent only to those neighbors that were of significant degree
 - ▶ Since there are fewer than K of these, simplify can then remove the coalesced node from the graph.
 - ▶ Thus if the original graph was colorable, the conservative coalescing strategy does not alter the colorability of the graph.